MutantChick: Type-Preserving Mutation Analysis for Coq

Matteo Cavada, Andrea Colò and Alberto Momigliano

DI, Università di Milano

Abstract. We present MutantChick, a mutation analysis tool for Coq, to be used in combination with QuickChick to evaluate the fault detection capability of property-based testing in a proof assistant. Mutation analysis of Coq theories is implemented via metaprogramming with MetaCoq and it is by construction type-preserving.

1 Introduction

Formal verification of software via (interactive) theorem proving gives the highest level of assurance, but it is still very labor-intensive. This effort may be furthermore wasted in the design phase of a software system, where mistakes and changes are likely to occur both at the level of the specification and of the implementation. Testing/validation/model-checking can ameliorate the situation by finding faults, in a mostly "push-button" way and help refining the spec/code until theorem proving can start. Even so, when do we stop testing and start proving? This is analogous to asking when to stop testing in the traditional software cycle and release the software itself. While taking into the right consideration Dijkstra's admonishments, in practical terms it much depends on how good our testing suite is.

In this paper we fix our validation technique to be property-based testing [10] (PBT), which tries to refute executable specifications against automatically generated data, typically in a pseudo-random way. Once the idea broke out in the functional programming community with QuickCheck, it spread to most programming languages and turned also in a commercial enterprise [12].

We can view PBT as a testing suite that consists of a set of properties plus a *data generation strategy*: while PBT tools vary in those strategies and in the amount of automation they offer, when dealing with random generation uniform distributions are rarely the most effective choice and the art of PBT resides in writing very sophisticated generators — see [11] for an intimidating example. But, how do we assess the fault detection capability of such a testing suite? If it stops finding new counterexamples, are we willing to consider the system-under-test ready for verification?

Mutation analysis [13] (MA) aims to evaluate software testing techniques with a form of white box testing, whereby a source program is changed in a localized way by introducing a single (syntactic) *fault*. The resulting program is called a "mutant" and hopefully it is also semantically different from its ancestor. A testing suite should recognize the faulted code, which is known as "killing" the mutant. The higher the number of killed mutants, the better the testing suite. While typically used in combination with unit testing, MA fits fairly well with PBT (as proposed in [17]) and, in fact, in the literature PBT tools are evaluated with manually inserted faults (e.g., the PLT-redex benchmark suite, see `http://docs.racket-lang.org/redex/benchmark.html`).

On the verification side, Coq (`https://coq.inria.fr`) is one of the leading proof assistants, which has shown its color both in formalizing general mathematics (see the proof of the Odd Order Theorem) and in specific domains such as the one dearest to our heart, namely the semantics of programming languages ([18]).

Coq is an extremely powerful logic, which accommodates programming, specification and proving in a constructive setting. In accordance with the Peter Parker principle "with great power comes great responsibility", PBT in such an environment is no small feat: QuickChick ([19]), while under active developments ([16,15]), brings in additional challenges as we touch upon in the next Section.

This is where MutantChick comes in: MutantChick is a mutation analysis tool for Coq theories, whose main application is evaluating the "oracle adequacy" of PBT with QuickChick. We accomplish this in a purely logical way via meta-programming with MetaCoq [20]. Since any MA tool is as good as the *mutation operators* it implements, we provide a DSL to specify and compute *type-preserving* mutation operators, supporting both the functional and the relational aspects of Coq: we also give the users the possibility to define their own. Differently from related tools ([17,4,8]), our guiding principle insists that operators are not just grammatical rules that injects faults, but must preserve types, so that any resultant mutant will type-check by construction; thanks to Coq's extremely strong type system, this also helps in reducing the combinatorial explosion intrinsic in operators application.

What MutantChick does not (yet) do is the full automation of the cycle of mutant generation, testing and scoring, in the sense that tools for imperative programming languages support ([13]). This is problematic, if one stays within the realm of a pure language such as (Meta)Coq.

We acknowledge that this short paper may be hard to read and therefore we have provided Appendix A and B with additional information. We assume some familiarity with Coq and the basics of PBT and refer to to the relevant chapter in *Software Foundations* `https://softwarefoundations.cis.upenn.edu/qc-current/` for QuickChick and to [6] for testing the case studies we mention. Further, we will use MetaCoq as a black box and will describe only some of the features of MutantChick without diving at all into its implementation. The tech report [3] and the repo (`https://bitbucket.org/matteo_cavada/tesi/src/master/mutantChick/`) give full details. The file `demo.v` therein offers a gentle introduction to MutantChick's main features.

2 Running Example

Our main interest is verification in a particular domain: the mechanization of the metatheory of programming languages (PL) and related calculi ([9]). As any practitioner may testify, the relevant properties are well known and have mathematically shallow proofs. The difficulty lies in the potential magnitude of the cases one must consider and in the trickiness that some encodings require. Here, very minor mistakes, even at the level of what we would consider a typo, may severely frustrate the verification effort, to the point to make it not cost-effective.

As a running example, we take a very simple model of a PL, a typed arithmetic language (`https://softwarefoundations.cis.upenn.edu/plf-current/Types.html`), featuring numerals with predecessor and Booleans with if-then-else and test-for-zero. For the sake of this Section, we concentrate on a rule-based presentation (akin to logic

programs) of the typing rules, but our approach covers the functional rendition as well. As a property, we consider the *progress* lemma, which ensures that if a term is well typed, then either it is a value or it can take a step; it is one of the building blocks in Milner's motto, for which a well-typed program cannot go wrong.

The typing rules are encoded as an **Inductive** definition `has_type` to be read "term `t` has type `T`" (see Appendix A for the other definitions):

```
Inductive has_type: tm → typ → Prop :=
  T_Tru: has_type ttrue TBool
| T_Fls: has_type tfalse TBool
| T_Test: ∀ t1 t2 t3 T, has_type t1 TBool → has_type t2 T → has_type t3 T →
    has_type (tif t1 t2 t3) T
| T_Zro: has_type tzero TNat
| T_Scc: ∀ t1, has_type t1 TNat → has_type (tsucc t1) TNat
| T_Prd: ∀ t1 has_type t1 TNat → has_type (tpred t1) TNat
| T_Iszro: ∀ t1, has_type t1 TNat → has_type (tiszero t1) TBool.

Theorem progress: ∀ t T, has_type t T → value t ∨ ∃ t', step t t'.
```

We now introduce some mutations that should mirror typical mistakes while designing a PL artifacts — see for example the manual mutations in the cited PLT-Redex benchmark suite. A first mutation [1] represents a simple typo where we have swapped `TBool` for `TNat` in the test-for-zero rule:

```
... | T_Iszro_M : ∀ t1 : tm, has_type_M1 t1 TBool → has_type_M1 (tiszero t1) TNat.
```

The second one corresponds to having forgotten an hypothesis in the rule for if-then-else, namely that the test `t1` is a Boolean:

```
... | T_Test_M: ∀ t1 t2 t3 T, has_type_M2 t2 T → has_type_M2 t3 T → has_type_M2 (tif
    t1 t2 t3) T
```

Finally, suppose instead we *add* an additional rule to the given ones (this is exercise 2 from `Types.html`):

```
... | T_SccB: ∀ t, has_type_M3 t TBool → has_type_M3 (tsucc t) TBool.
```

All mutants do *not* satisfy progress: it would stand to reason that a PBT tool should find the relevant counterexamples, as say αCheck does quite easily and without any complicated setup [5]. However, this is not immediately the case with QuickChick. Coq is, under the Curry-Howard view, a version of constructive higher-order logic, enriched with (co)inductive types and universe polymorphism, where only pure total functional programs are permitted, while allowing highly undecidable specifications. This duality is reflected in the type Set of computations and Prop of arbitrary propositions. If we wish to test a conjecture, it must be shown effectively computable, in QuickChick's terms it is a Checkable property. There are two ways to go about it: first, if the notion under test is defined relationally, that is at Prop, we can manually provide a proof that it is indeed decidable. In certain simple cases such as the `has_type` relation, we can rely on the rather unpredictable tool for automatic derivations of generators out of **Inductive** definitions [16]. The second and fail-safe way is to translate every spec into a Boolean-valued function: since in Coq every such function must be total, decidability and therefore Checkability is guaranteed. This approach has its

[1] Each mutation yields a new **Inductive** definition named by convention `has_type_Mn`.

drawbacks, since it needs to explicitly address the partiality of rule-based specifications and to accommodate Coq's rather temperamental totality checker. Still, even stating the `progress` property so that it can be tested is not immediate, neither with automatic derivation (**Definition** `progressST`), nor with a custom-made generator (**Definition** `progressGen`), see Appendix A and [6].

All these layers point to the utility of a MA tool to assess the "oracle adequacy" of QuickChick. In the next Section we will see how MutantChick can derive those mutations. Note that in our example properties (viz. progress) are *trusted*. This is generally the case for PL semantics, since they come from the theorems that should hold for the underlying calculus; however, a large number of surviving mutants may also suggest a possible under-specification in such properties or a fault in its testable implementations. Finally, since properties are just Coq terms, our tool could mutate them as well, although we do not see this as particularly useful.

3 MutantChick

Our tool is based on MetaCoq [20], a library that provides rich meta-programming features for Coq. Among those, we will use 1) the (anti)quotation mechanism: quoting a Coq term `t`, denoted by `<% t %>`, induces a bijection between Coq kernel terms and a deep embedding in Coq itself of the syntax of the terms of the underlying Calculus of (Co)Inductive Constructions; the embedding is realized as an **Inductive** definition of the type `term`; 2) a reification of the static and dynamic semantics of the representation of the calculus, providing a hook for checking well-typedness of quoted terms.

For example, quoting the lambda expression **fun** `x` ⇒ `2 + hd x [9;8;7]` that sums 2 to the head of a given list returns the following abstract syntax tree (AST), which we have significantly abridged by using a number such as 2 to stand for its quotation — keep in mind that in Coq naturals and lists are not built-in, but obtained as inductive types.

```
(tLambda (nNamed "x")
   (tInd {| inductive_mind := "Coq.Init.Datatypes.nat"; inductive_ind := 0 |} [])
   (tApp (tConst "Coq.Init.Nat.add" [])
      [2; tApp (tConst "Coq.Lists.List.hd" [])
         [tInd {| inductive_mind := "Coq.Init.Datatypes.nat"; inductive_ind := 0 |}
              [];
          tRel 0;
          tApp (tConstruct {| inductive_mind := "Coq.Init.Datatypes.list";
               inductive_ind := 0 |} 1 [])
            [tInd {| inductive_mind := "Coq.Init.Datatypes.nat"; inductive_ind := 0
                 |} [];
             tApp (tConstruct {| inductive_mind := "Coq.Init.Datatypes.nat";
                  inductive_ind := 0 |} 1 []) [9;8;7]]]]))
```

To access and modify Coq's terms and environment, MetaCoq provides a monad (`TemplateMonad`) in which it is possible (among other things) to describe quoting and unquoting actions of Coq terms, as well as other (few) actions with side-effects, such as raising an exception and inserting new terms inside Coq's environment; these monadic meta-programs, once invoked with the `Run TemplateProgram` command, will be executed by an interpreter inside the OCaml environment.

Kind	Operator	Description
General	Substitute	Substitutes a term with another
	Swap	Exchanges two terms
	IfThenElse	Exchanges then and else branch
	DelImplications	Deletes assumption
	UserDefined	User defined operator
Inductive	NewConstructor	Adds new constructor
	SubConstructor	Substitutes a constructor
	OnConstructor	Maps an operator over a constructor
	OnConstructors	Maps an operator over an inductive def.

Table 1. List of MutantChick's operators.

MutantChick aims to devise *mutation operators* for a proof assistant, here Coq (but they would also be relevant for other systems such as *Lean* or *Isabelle/HOL*), which embodies both functional and relational specifications, with a particular attention to our intended domain of application. Those operators specify the mutations that the tool will inject, trying to simulate natural occurring bugs. This is justified by the "competent programmer assumption" [13], according to which the latter tends to develop programs close to the correct version and thus the difference between current and correct code for each fault is small.

An MA tool is as effective as its operators are relevant. Historically, MA comes from imperative languages and the operators thereby still owe to those initially devised for FORTRAN. Even operators proposed for declarative programs ([8,17]) make only partial sense, being untyped and too linked to the operational semantics of the programming language. Usable examples comprise arithmetic and relational operator mutation, say turn $\times$ into $+$ and $<$ into $\leq$ and constant mutations, say 0 into 1.

Table 3 describes, at a high level, the operators currently implemented in MutantChick. We have two (non-disjoint) main categories: *general* operators, working on any Coq term, and *inductive* ones, which are particularly suited to relationally defined judgments.

The general syntax to mutate a terms is as follow (see Appendix B for the BNF of the DSL):

```
Run TemplateProgram (
    Mutate <% term_to_mutate %>
           using (operator)
           named "new_name").
```

Let us see a classic arithmetic mutation: we can use simple substitution `t1 ==> t2` to substitute every occurrence of `t1` with `t2`. This operator is internally defined as an inductive definition `Substitute {A} (t1 t2: A)`, which is indexed by the type `A` of the terms in order to ensure that the substitution is type-preserving. For example:

```
Run TemplateProgram (
  Mutate <% 4 * 5 %>
         using (Init.Nat.mul ==> Init.Nat.add)
         named "out").
Print out.
```

```
out = 4 + 5
    : nat
```

All the generic operators works not only on plain Coq terms, but also on **Definition**, **Fixpoint**, **Theorem** etc.

Since a quoted term is a (rather large) AST, a mutation operator is implemented by two components: 1) a *search phase*, in which we find *all* locations inside the tree of the to-be-mutated term; this is accomplished by a function `check :: term → bool` that establishes if a given quoted term is suitable for mutation; 2) a *mutation phase*, in which a transformation function `trans :: term → term` is applied to the terms in the locations previously identified.

An important optimization in MA is *random generation* of mutants[13]: since a given mutation can be applied multiple time in the same term, it is reasonable to select only a sample thereof. Again, a pure language such as Coq cannot rely on an external pseudo-random number generator, hence we had to deploy our own as a *Linear-Feedback Shift Register* on 7 bits; this generates pseudo-random sequences of numbers starting from a seed (at this stage provided by the user).

The DSL offers three choices: 1) *mutate everywhere*; 2) *pick mutations with probability p*, i.e., generate a random number for every mutable location in the AST: if it is bigger then a certain threshold (chosen by the user), then collect the mutant; 3) *pick a maximum of n mutations*, i.e., maximum of n location are drawn among all the coordinates found in the *search* phase.

Rather than seeing each operator in details (for which we refer once again to [3]), we detail how MutantChick realizes the mutations of the previous Section.

For the typo, we use the bidirectional substitution `Swap t1 t2`; it exchanges the order of appearance of `t1` and `t2` inside an expression and is defined in term of `Substitute`. Note that `t1` and `t2` can be (bound) variables, which mirrors a large category of errors.

```
Run TemplateProgram (
  Mutate <% ∀ t1, has_type t1 TNat → has_type (tiszero t1) TBool. %>
         using (Swap <% TBool %> <% TNat %>)
         named "T_Iszro_M").
```

We could have achieved this and the next mutation with the operator `SubConstructor` although with less automation.

`DelImplications` mirrors missing assumptions by non-deterministically removing implications (technically, dependent products); in the following case it will yield 6 different non-isomorphic mutants.

```
Run TemplateProgram (
  Mutate <% ∀ t1 t2 t3 T, has_type t1 TBool → has_type t2 T → has_type t3 T →
      has_type (tif t1 t2 t3) T %>
        using (DelImplications)
        named "T_Test_M").
```

Interestingly, we can make this require less interaction by using the `OnConstructors` combinator that maps the operator over the whole relation; since this could generate a lot of mutants, we use random choice to select a few.

```
Run TemplateProgram (
  Run TemplateProgram (
       Mutate <% has_type %>
              using (OnConstructors DelImplications)
              named "has_type_mut"
              generating (RandomMutations 34 64)).
```

Finally, we can add a new constructor to an **Inductive** definition via `NewConstructor`

```
Run TemplateProgram (
 Mutate <% has_type %>
        using (NewConstructor "T_SccB"
              <% fun hastype_M1 ⇒ ∀ t, has_type_M1 t TBool →
                   has_type_M1 (tsucc t) TBool %>)
         named "hastype_M1").
```

4 Conclusions, related and future work

In this short paper we have sketched the rationale and the high level design of MutantChick, a mutation analysis tool for the Coq proof assistant. We have advocated the usefulness of such a tool, given the popularity that testing, in particular PBT, is having in the Coq world, in terms of applications ([11,1]) and development ([16,15]). Still, mutation *testing*, by which we mean the *improvement* of a testing suite via MA, has been lagging behind, being only informally and manually used to evaluate PBT. The exception is `mCoq` [4], which follows a very different philosophy from ours: it exports the AST of a Coq term as an *S-expression* thanks to an OCaml serializer and then applies mutations to the sexp using a tool written in Java. Following this choice, it cannot enforce that mutation operators are type-preserving, nor can they be easily extended by the user. In fact, the hardwired operators are quite limited, being mostly concerned with functional programming. We also do not share `mCoq`'s motivation, which are not PBT but *mutation proving*, whereas mutations are introduced in definitions to see if the proof scripts of related theorems still hold: if they do, it means that those definitions are under-specified, if not unused. It seems to us a very big hammer for such a small nail.

We have evaluated the tool on two case studies ([6,3]): the arithmetic language and much more significantly, Leroy & Appel's *list-machine* benchmark [2]. In both, we have realized the mutations described in the literature [14], implemented the relevant generators and tested various forms of type soundness with QuickChick. The list-machine case study is particularly meaningful: the properties under test have extremely sparse precondition, which forces us to implement complex generators with intricated heuristics built-in, whose fault detection capability which we would be hard to gauge without a MA tool.

The main limitation of MutantChick is performances: mutation analysis is computationally expensive and (Meta)Coq is definitely not an efficient general-purpose programming language, neither w.rt. memory consumption nor the boilerplate required

to generate and test mutants in a completely automated way. We have managed to use operators such as `==>` to percolate mutated terms within all the definitions that depend on them in a Coq theory, but to address the whole mutation cycle fully, we would need, among others, to read and write files, which is out of Coq's remit.

To address performances, a possibility is the *extraction* to OCaml of the MetaCoq code responsible for the heaviest computations: this is possible but not trivial (see Sec. 5 of [20]). A more radical solution would be to port MutantChick to Elpi [7], the λProlog interpreter embedded inside Coq, and exploit full-range logic programming.

References

1. D. Annenkov, M. Milo, J. B. Nielsen, and B. Spitters. Verifying, testing and running smart contracts in concert.
2. A. W. Appel, R. Dockins, and X. Leroy. A list-machine benchmark for mechanized metatheory. *J. Autom. Reasoning*, 49(3):453–491, 2012.
3. M. Cavada. Property-based testing and mutation analysis in Coq. Technical report, DI, University of Milano, 2020. http://dx.doi.org/10.13140/RG.2.2.18766.69445.
4. A. Çelik, K. Palmskog, M. Parovic, E. J. G. Arias, and M. Gligoric. Mutation analysis for coq. In *ASE*, pages 539–551. IEEE, 2019.
5. J. Cheney and A. Momigliano. αCheck: A mechanized metatheory model checker. *TPLP*, 17(3):311–352, 2017.
6. A. Colò. Property-based testing of the list-machine benchmark with QuickChick. Technical report, DI, University of Milano, 2020. 10.13140/RG.2.2.19283.60962.
7. C. Dunchev, F. Guidi, C. Sacerdoti Coen, and E. Tassi. ELPI: Fast, Embeddable, λProlog Interpreter. In *Proceedings of LPAR*, Suva, Fiji, Nov. 2015.
8. A. Efremidis, J. Schmidt, S. Krings, and P. Körner. Measuring coverage of prolog programs using mutation testing. *CoRR*, abs/1808.07725, 2018.
9. M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.
10. G. Fink and M. Bishop. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, pages 74–80, July 1997.
11. C. Hritcu, L. Lampropoulos, A. Spector, A. de Amorim, M. Dénès, J. Hughes, B. Pierce, and D. Vytiniotis. Testing noninterference, quickly. *J. Funct. Program.*, 26:e4, 2016.
12. J. Hughes. Quickcheck testing for fun and profit. In *PADL'07*, LNCS, pages 1–32, Berlin, Heidelberg, 2007. Springer-Verlag.
13. Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, Sept. 2011.
14. F. Komauli and A. Momigliano. Property-based testing of the meta-theory of abstract machines: an experience report. In *CILC*, volume 2214 of *CEUR*, pages 22–39, 2018.
15. L. Lampropoulos, M. Hicks, and B. C. Pierce. Coverage guided, property based testing. *Proc. ACM Program. Lang.*, 3(OOPSLA):181:1–181:29, 2019.
16. L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce. Generating good generators for inductive relations. *POPL*, pages 1–30, 2017.
17. D. Le, M. A. Alipour, R. Gopinath, and A. Groce. MuCheck: an extensible tool for mutation testing of Haskell programs. In *ISSTA*, pages 429–432. ACM, 2014.
18. X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.
19. Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. In *ITP*, pages 325–343. Springer, 2015.
20. M. Sozeau, A. Anand, S. Boulier, C. Cohen, Y. Forster, F. Kunze, G. Malecha, N. Tabareau, and T. Winterhalter. The MetaCoq project. *J. Autom. Reasoning*, pages 1–53, 2020.

A Listings of the Typed Arithmetic Language

In this section we list the code we have selectively quoted in Section 2. For the sake of testing the dynamic semantics is encoded functionally (so that it is trivially a decidable `boolean` property).

```
Inductive tm : Type := | ttrue : tm | tfalse : tm | tif : tm → tm → tm → tm
 | tzero : tm | tsucc : tm → tm | tpred : tm → tm | tiszero : tm → tm.

Inductive typ : Type := | TBool : typ |TNat : typ.

Fixpoint isnumericval (t:tm) : bool :=
 match t with
  | tzero ⇒ true
  | tsucc t1 ⇒ isnumericval t1
  | _ ⇒ false
 end.

Fixpoint isval (t:tm) : bool :=
 match t with
  | ttrue ⇒ true
  | tfalse ⇒ true
  | t ⇒ isnumericval t
  end.

Fixpoint stepF (t:tm) : option tm :=
 match t with
  | tif ttrue t2 t3 ⇒ ret t2
  | tif tfalse t2 t3 ⇒ ret t3
  | tif t1 t2 t3 ⇒
     t1' ← stepF t1 ;;
     ret (tif t1' t2 t3)
  | tsucc t1 ⇒
     t1' ← stepF t1 ;;
     ret (tsucc(t1'))
  | tpred tzero ⇒ ret tzero
  | tpred (tsucc nv1) ⇒
     if (isnumericval nv1) then ret nv1 else
     t1' ← stepF nv1 ;;
     ret (tpred(tsucc t1'))
  | tpred t1 ⇒
     t1' ← stepF t1 ;;
     ret (tpred t1')
  | tiszero tzero ⇒ ret ttrue
  | tiszero(tsucc nv1) ⇒
     if (isnumericval nv1) then ret tfalse else
     t1' ← stepF nv1 ;;
     ret (tiszero(tsucc t1'))
  | tiszero t1 ⇒
     t1' ← stepF t1 ;;
     ret (tiszero t1')
  |_ ⇒ None
 end.

Definition canStep (e:tm) : bool :=
 match stepF e with
 | Some _ ⇒ true
 | None ⇒ false
 end.
```

Next, we list the code of a generator for well-typed terms of size n, which heavily relies on QuickChick's generators combinators:

```
Fixpoint gen_term_size (n:nat) (t:typ) : G tm :=
 match n with
  | 0 ⇒
```

```
  match t with
  | TNat ⇒ returnGen tzero
  | TBool ⇒ oneOf [returnGen ttrue; returnGen tfalse]
  end
| S n' ⇒
  m ← choose (0, n');;
  match t with
  | TNat ⇒
   oneOf [returnGen tzero;
       liftGen tsucc (gen_term_size (n'−m) TNat) ;
       liftGen tpred (gen_term_size (n'−m) TNat) ;
       liftGen3 tif (gen_term_size (n'−m) TBool)
              (gen_term_size (n'−m) TNat)
              (gen_term_size (n'−m) TNat)]
  | TBool ⇒
   oneOf [returnGen ttrue; returnGen tfalse;
      liftGen tiszero (gen_term_size (n'−m) TNat);
      liftGen3 tif (gen_term_size (n'−m) TBool)
              (gen_term_size (n'−m) TBool)
              (gen_term_size (n'−m) TBool)]
  end
end.
```

To give a rough idea on how properties are encoded, we report two *Checkable* definitions of progress, the first using the custom generator `gen_term`, the second based on automatic derivation of generators for types `forAll arbitrary (fun tau : typ ⇒ ...)` and of well typed terms out of the **Inductive** definition, namely `forAll ( genST (fun t ⇒ has_type t tau))`. For a full explanation, please refer to [6].

```
Definition progressGen :=
 forAll arbitrary (
     fun tau ⇒
      forAll (gen_term tau)
       (fun t ⇒
       isval t || canStep t)).
```

```
Definition progressST :=
 forAll arbitrary (fun tau ⇒
 forAll (genST (fun t ⇒ has_type t
      tau))
     (fun mt ⇒
      match mt with
      | Some t ⇒ (isval t || canStep t)
      | None ⇒ false
      end)).
```

B MutantChick's DSL

A BNF grammar of the DSL and of the operators is as follows:

⟨*root*⟩ ::= '`Mutate`' ⟨*MetaTerm*⟩ '`using`' ⟨*op*⟩ '`named`' ⟨*string*⟩ {⟨*random*⟩}
  | '`MultipleMutate`' ⟨*MetaTerm*⟩ '`using`' ⟨*opList*⟩ '`named`' ⟨*string*⟩ {⟨*random*⟩}

⟨*random*⟩ ::= '`generating`' ⟨*randomConfig*⟩

⟨*randomConfig*⟩ ::= '`AllMutations`'
  | '`RandomMutations`' ⟨*nat*⟩ ⟨*nat*⟩

⟨*string*⟩ ::= A Coq string

⟨*checkFunction*⟩ ::= A Coq function of type `term → bool`

⟨*transFunction*⟩ ::= A Coq function of type `term → term`

⟨*coqExpr*⟩ ::= Any Coq expression

⟨*MetaTerm*⟩ ::= '`<%`' ⟨*coqExpr*⟩ '`%>`'

⟨*op*⟩ ::= ⟨*coqExpr*⟩ '`==>`' ⟨*coqExpr*⟩
| '`Swap`' (⟨*string*⟩ | ⟨*metaTerm*⟩) (⟨*string*⟩ | ⟨*metaTerm*⟩)
| '`IfThenElse`'
| '`DelImplications`'
| '`NewConstructor`' ⟨*string*⟩ ⟨*metaTerm*⟩
| '`SubConstructor`' ⟨*string*⟩ ⟨*string*⟩ ⟨*metaTerm*⟩
| '`OnConstructor`' ⟨*string*⟩ ⟨*op*⟩
| '`OnConstructors`' ⟨*op*⟩
| '`UserDefined`' ⟨*checkFunction*⟩ ⟨*transFunction*⟩

`Mutate` takes four arguments: the expression to mutate, the operator, the base-name for the newly generated definitions and an optional arguments indicating whether to use random selection or not. `MultipleMutate` is identical to `Mutate`, but it accepts a *list* of operators.